Nikke Vartia

Serialisointiliitännäinen Unity-pelikehitysympäristöön

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

8.11.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Nikke Vartia Serialisointiliitännäinen Unity-pelikehitysympäristöön 20 sivua 8.11.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Miikka Mäki-Uuro

Insinöörityön tavoitteena oli toteuttaa serialisointiliitännäinen Unity-pelikehitysympäristöön, jonka avulla voidaan tallentaa Unitylla luodun pelin tila tiedostoon ja ladata tallennettu tila tiedostosta takaisin peliin. Liitännäisen päävaatimuksina oli monialustaisuus ja peliobjektien sijainnin serialisointi. Lisäksi liitännäistä luodessa panostettiin sen helppokäyttöisyyteen ja yksinkertaisuuteen, koska tavoitteena oli myös julkaista liitännäinen Unity Asset Storessa.

Aluksi työssä esitellään Unity-pelikehitysympäristö ja erilaisia serialisointiformaatteja. Liitännäisessä päädyttiin käyttämään tekstityyppistä JSON-serialisaatiota. Itse liitännäinen toteutettiin C#-ohjelmointikielellä. Ongelmallisinta serialisoinnin kannalta oli Component-tyypin serialisointi. Se ratkaistiin käyttämällä reflektiota.

Työn lopussa esitellään toteutuksessa huomattuja rajoitteita ja pohditaan, minkälaista jatkokehitystä voisi vielä tehdä. Liitännäisen todettiin pääosin täyttävän asetetut vaatimukset. Helppokäyttöisyyden puutteiden vuoksi liitännäisen ei koettu olevan vielä valmis Asset Storeen.

Avainsanat	Unity, serialisointi, C#, .NET, JSON

Author(s) Title Number of Pages Date	Nikke Vartia Serialization Plugin for Unity 20 pages 8 November 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Miikka Mäki-Uuro, Senior Lecturer

The goal of this thesis was to create a serialization plugin for Unity allowing users to save the current state of the game into a file and vice versa. The main requirements for the plugin were to support multiple platforms and the ability to serialize the position of game objects. Ease of use and simplicity were kept in mind while developing the plugin as releasing the plugin on Unity Asset Store was an additional goal.

The thesis begins with an overview of Unity and various serialization formats. The text based JSON format was chosen as the serialization format for the plugin. The plugin itself was written in the C# programming language. The biggest hurdle was figuring out how to serialize all the components of a game object without specifying how each component should be serialized. That was resolved with reflection.

Multiple minor shortcomings were noted and the thesis provides suggestions as how to overcome them. The conclusion was that the plugin met its requirements for the most part. There were some shortcomings regarding the ease of use, and therefore the plugin was not yet released on Asset Store.

Keywords	Unity, serialization, C#, .NET, JSON

Sisällys

1 Johdanto

Pelien suosio ja peliteollisuus ovat jatkuvassa nousussa. Vauhdikkaimmassa nousussa on mobiilipeliteollisuus. Vuonna 2016 mobiilipelit tulevat kattamaan arviolta jopa 37 % maailman pelimarkkinoistatuloista [1]. Tuhannesta suosituimmasta ilmaisesta mobiilipelistä 34 % on tehty käyttäen Unity-pelikehitysympäristöä. Unity on siis suosituin kolmannen osapuolen pelikehitysympäristö ja sillä on toteutettu lukuisia menestyksekkäitä pelejä [2].

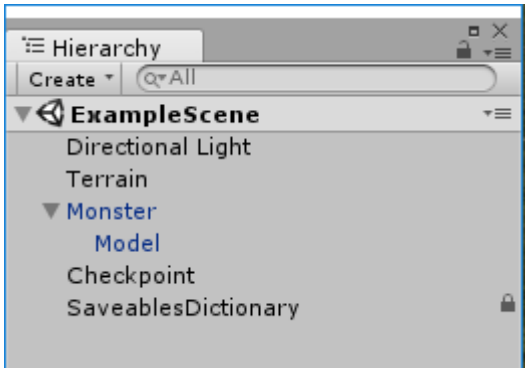
Lähes jokaisessa pelissä tallennetaan käyttäjän edistys joko tietokantaan tai tiedostoon. Unity ei kuitenkaan tarjoa suoraan valmista palikkaa tilan tallentamiseen. Unityn Asset Storesta löytyy valmiita tallennusliitännäisiä. Osat niistä ovat maksullisia, osat eivät tue peliobjektien sijainnin tallennusta ja osat vaikuttavat sopivilta, mutta ne on luotu Unityn aikaisemmille versioille eivätkä ole siten enää tuettuja. Koska Asset Storesta ei löydy ilmaista liitännäistä, joka vastaa tarpeita, sellainen toteutetaan itse.

Insinöörityön tarkoituksena on toteuttaa serialisointiliitännäinen Unity-pelikehitysympäristöön, joka voidaan julkaista Unityn Asset Storessa. Liitännäisen avulla käyttäjä voi tallentaa pelin tilan tiedostoon ja ladata aikaisemmin tallennetun tilan. Koska tavoitteena on julkaista liitännäinen Asset Storessa, sitä ei voi luoda vastaamaan suoraan omia tarpeita, vaan siitä on tehtävä sen verran yleiskäyttöinen, että muutkin hyötyvät liitännäisen käytöstä. Liitännäinen ohjelmoidaan käyttäen C#-ohjelmointikieltä.

2 Unity

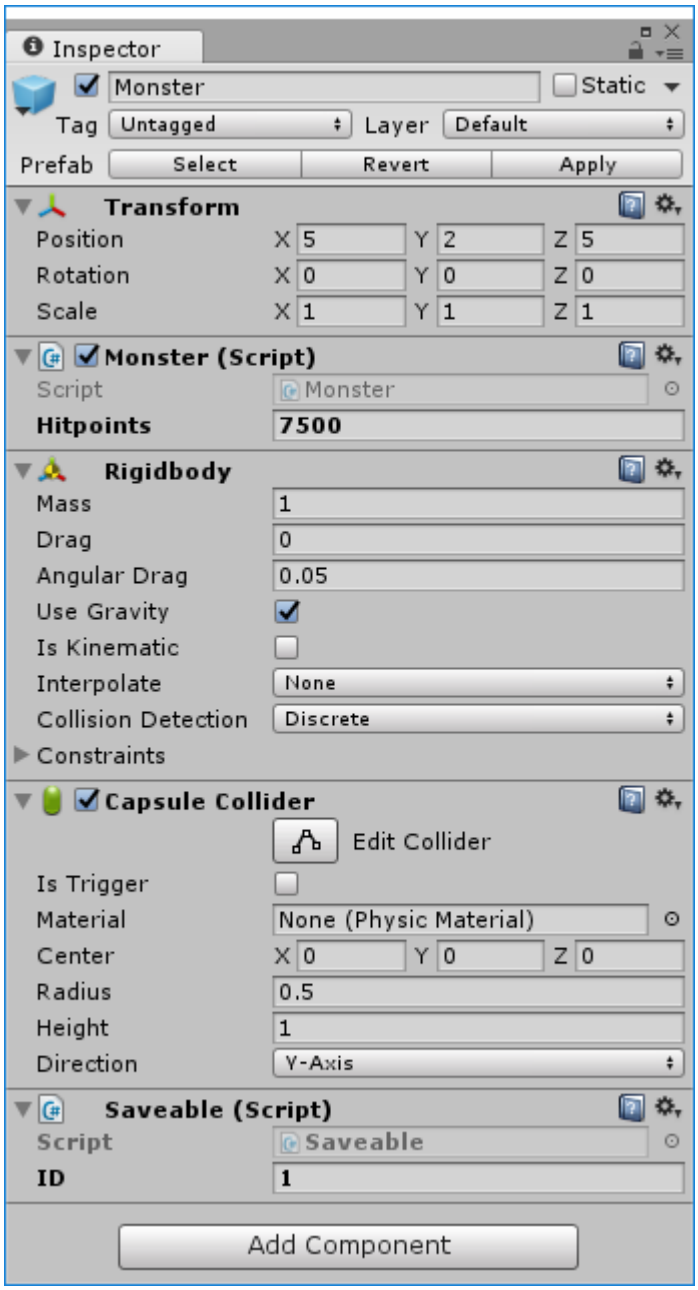
Unity on vuonna 2005 julkaistu Unity Technolgiesin kehittämä pelikehitysympäristö, jolla pystyy kehittämään sekä kaksi- että kolmiulotteisia pelejä yli kahdellekymmenelle eri alustalle [3; 4]. Unity on erittäin suosittu ja sitä kehitetään edelleen aktiivisesti [2]. Tällä hetkellä Unityssa voi valita käytettäväksi laitteistoriippumatonta OpenGL-grafiikkarajapintaa ja esimerkiksi Windows-alustalle tarjolla on myös Direct3D9, Direct3D11 ja Direct3D12. Lisäksi Vulkan-tuki on parhaillaan kehitteillä [5]. Fysiikan mallinnukseen Unity käyttää PhysX 3.3 -fysiikkamoottoria [6]. Unityn pelimoottori on luotu käyttäen C- ja C++ -ohjelmointikieliä. Käyttäjille vaihtoehtoina ovat skriptikielet C#, UnityScript tai Boo. [7.]

Unitylla luodut pelit koostuvat yhdestä tai useammasta Scenestä. Tyypillisiä Scenejä ovat esimerkiksi päävalikko ja pelin yksittäiset kentät. Scenet pitävät sisällään siihen kuuluvat peliobjektit (GameObject), kuten pelihahmot, UI-elementit, tasanteet ja valot. Peliobjekteilla voi olla myös lapsipeliobjekteja. Kun peliobjekti liikkuu pelimaailmassa, kaikki lapsipeliobjektit liikkuvat sen mukana. Kuvassa 1 näkyy pelikehitysympäristön hierarkianäkymä. Näkymässä näkyy ladattu Scene ja kaikki Scenessä olevat peliobjektit.



Kuva 1. Pelikehitysympäristön hierarkianäkymä

Yksittäiseen peliobjektiin voi olla liitettynä useita komponentteja, jotka määrittävät peliobjektin ominaisuudet ja toiminnallisuuden. Unity tarjoaa valmiita komponentteja peliobjektin toiminnallisuuden ja ulkoasun toteuttamiseen. Käyttäjä pystyy viittaamaan ja ohjaamaan näitä komponentteja käyttämällä niiden rajapintaa skripteistä käsin. Jokaisessa peliobjektissa on vähintään Transform-komponentti, joka ilmaisee peliobjektin sijainnin, kierron ja skaalan. Kuvassa 2 on pelikehitysympäristön Inspector-näkymä, jossa näkyy ExampleScenessä olevan Monster-peliobjektin komponentit.



Kuva 2. Inspector-näkymä

Peliobjektikokonaisuuksista on mahdollista luoda .prefab-päätteisiä tiedostoja. Prefabit ovat peliobjektimalleja, jotka nopeuttavat samanlaisten peliobjektien lisäämistä Sceneen. Prefabit näkyvät hierarkiassa sinisinä ja prefabista eroavat arvot Inpector-näkymässä tummennettuina.

Microsoftin kehittämä .NET-ohjelmistokomponenttikirjasto, jonka päälle C#-ohjelmointikieli on kehitetty, tarjoaa valmiin rajapinnan luokkien serialisointiin XML- ja binäärimuo-

dossa. Nämä rajapinnat yrittävät serialisoida kaikki luokan julkiset kentät. Unity ei kuitenkaan tue pelimoottorityyppien, kuten Transform-komponentin, serialisointia ja aiheuttaa poikkeuksen. Tämä vaikeuttaa kaikkien komponenttien serialisointia, sillä kaikissa komponenteissa on viittaus peliobjektiin, johon komponentti on liitettynä, ja kaikissa peliobjekteissa on viittaus sen Transform-komponenttiin. Koska nämä viittaukset ovat julkisia kenttiä, komponentin serialisointi yrittää myös serialisoida pelimoottorityyppejä.

Unity tukee kahta erityyppistä liitännäistä: hallitut liitännäiset (Managed Plugins) ja natiivit liitännäiset (Native Plugins). Hallitut liitännäiset koostuvat pelkästään .NET-ohjelmistokomponenttikirjastoon pohjautuvasta koodista. Tästä johtuen hallitut liitännäiset eivät juurikaan eroa muista Unity-skripteistä, ja Unity kykenee kääntämään liitännäisen kaikille Unityn tukemille alustoille.

Natiivit liitännäiset ovat alustakohtaisia liitännäisiä. Ne on luotu käyttäen C-pohjaista ohjelmointikieltä. Natiiveilla liitännäisillä pystyy toimimaan .NET-ohjelmistokomponenttikirjaston ulkopuolella, kuten tekemään C:n järjestelmäkutsuja tai käyttämään kolmansien osapuolien luomia kirjastoja. Natiivisista liitännäisistä joutuu kuitenkin luomaan oman kirjastotiedoston jokaista alustaa kohden, jota liitännäisen halutaan tukevan.

Asset Store on Unity Technologiesin ylläpitämä palvelu, jossa käyttäjät voivat myydä tai jakaa ilmaiseksi kehittämiään paketteja (asset). Asset on kokoelma Unityn tukemia tiedostoja. Näitä assetteja ovat esimerkiksi ohjelmointikirjastot tai liitännäiset, kolmiulotteiset mallit, partikkeliefektit ja kehitysympäristölaajennukset.

3 Serialisointi

Serialisointi on prosessi, jossa objekti muutetaan sarjaksi tavuja, jotta objekti voidaan myöhemmin luoda uudestaan tai palauttaa aikaisempaan tilaan. Yleensä serialisoidut objektit tallennetaan tietokantaan tai tiedostoon tai lähetetään verkon yli toiseen laitteeseen. Tavujen muuntamista takaisin objektiksi kutsutaan deserialisoinniksi. [8.]

Serialisointiformaatit voidaan jaotella binääri- ja tekstiformaatteihin. Binääriformaattiin tallennettaessa on vaikeaa turvata deserialisoinnin onnistuminen eri alustalla kuin millä serialisointi tapahtuu.

Tekstimuotoisen serialisoinnin etuna ovat luettavuus ja monialustaisuus. Yleisin serialisaatioformaatti Extensible Markup Language (XML). Se on merkintäkieli ja sitä yleisimmin käytetään erilaisten dokumenttien tallentamiseen, mutta sen avulla voidaan myös esittää tietorakenteita tekstimuotoisena. Toinen yleinen tapa esittää tietorakenteita tekstimuotoisena on JavaScript Object Notation (JSON). JSON on XML:ää yksinkertaisempi ja kevyempi, ja se soveltuu paremmin tietorakenteiden kuvaamiseen. [9.]

.NET-ohjelmistokomponenttikirjasto tarjoaa serialisointia varten muun muassa System.Xml.Serialization- ja System.Runtime.Serialization-nimiavaruudet. System.Runtime.Serialization-nimiavaruudesta löytyy BinaryFormatter-luokka, jonka avulla objekteja voi serialisoida binääriformaattiin. Vastaavasti System.Xml.Serialization-nimiavaruuden avulla voi serialisoida objekteja XML-formaattiin. Näiden serialisointitapojen suorituskyky vaihtelee paljon serialisoitavan datan perusteella [10].

Lisäksi Unity tarjoaa JsonUtility-luokan, jonka avulla objekteja voi serialisoida JSON-formaattiin. JsonUtility on muita JSON-serialisointikirjastoja nopeampi, erityisesti deserialisoitaessa. Tämä johtunee JsonUtilityn yksinkertaisuudesta ja samalla sen vähäisistä ominaisuuksista. [11.]

4 Liitännäisen suunnittelu

4.1 Liitännäisen ominaisuudet

Yksi liitännäisen päävaatimuksista on monialustaisuus, koska Unitylla on mahdollista luoda peli samanaikaisesti useammalle eri alustalle. On esimerkiksi mahdollista, että liitännäisen luoma tiedosto pelin tilasta lähetetään pilvipalvelimelle yhdellä alustalla ja ladataan pilvipalvelimelta toisella alustalla, joten serialisoinnin ja deserialisoinnin on tapahduttava samalla tavalla alustojen kesken.

Toinen päävaatimus on peliobjektien Transform-komponentin kenttien tallennus. Jo pelkän Transform-komponentin tallennuksen avulla saa esimerkiksi pelaajan ja vihollisten sijainnin. Peliobjektien tilasta jää kuitenkin paljon tallentamatta, kuten esimerkiksi pelaajan ja vihollisten elämäpisteet. Lisätavoitteina ovat helppokäyttöisyys, yksinkertaisuus ja mahdollisuus Transform-komponentin serialisoinnin lisäksi serialisoida myös muita komponentteja, mukaan lukien käyttäjän itse luomat komponentit.

Monialustaisuusvaatimuksen takia serialisointityypiksi valittiin tekstimuotoinen tiedostoformaatti. Tekstimuotoisuus takaa sen, että serialisointi ja deserialisointi ovat yhtenäisiä eri alustojen välillä. Koska liitännäisen tarkoituksena on kuvata objektien sisältämää tietoa, valittiin serialisointiformaatiksi JSON. Lisäperusteluna JSON-formaatille on sen keveys.

Käyttäjän on helposti pystyttävä määrittelemään, minkä tyyppiset peliobjektit serialisoidaan. Tätä varten luodaan näkymä kehitysympäristöön, josta käyttäjä voi valita yhden tai useamman peliobjekti-prefabin, joiden pohjalta luotuihin peliobjekteihin lisätään tunnistenumero. Lisäksi käyttöliittymästä voi muokata luotavan tiedoston tiedostopolkua ja nähdä, mitkä peliobjektit tullaan serialisoimaan.

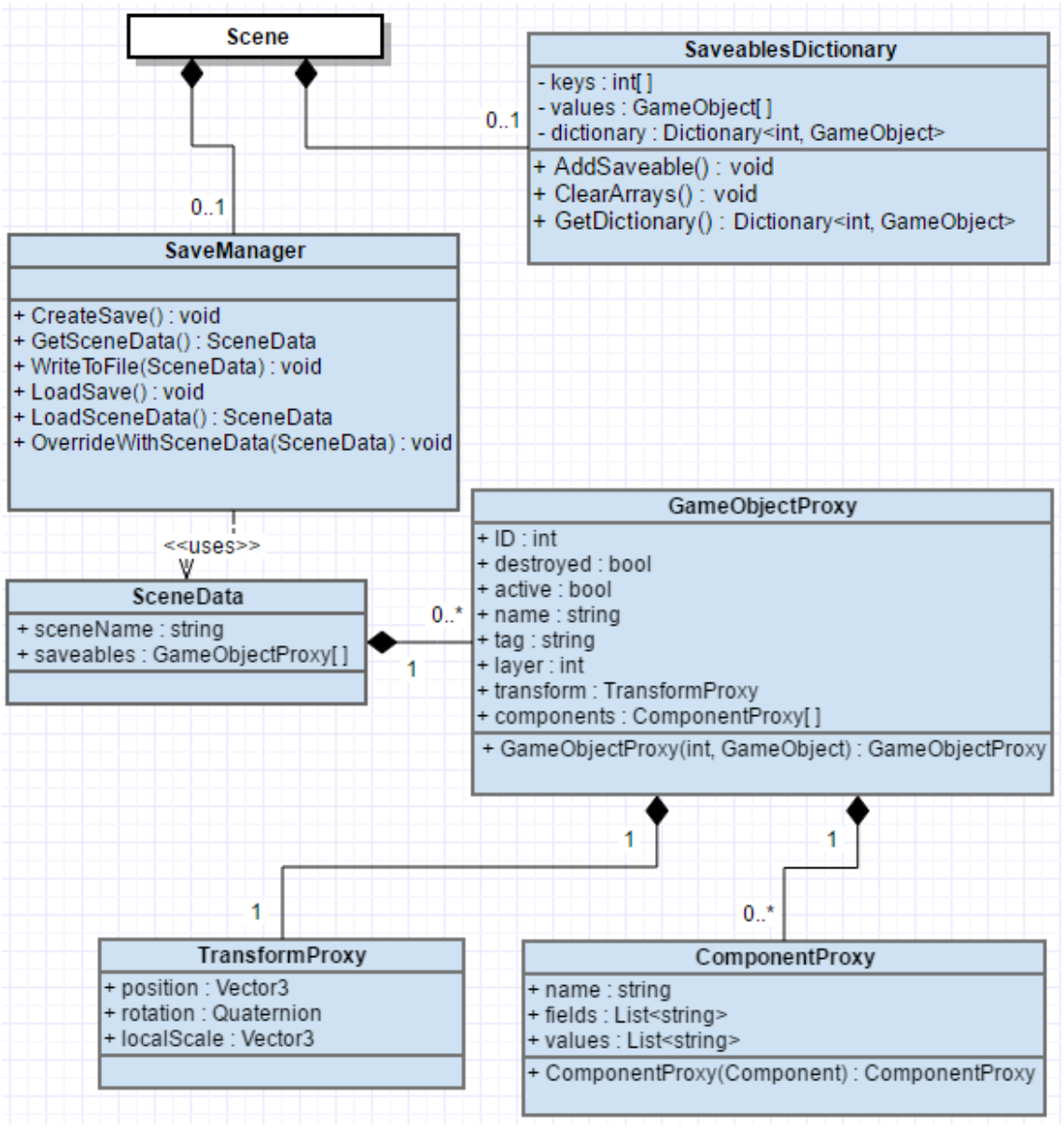
4.2 Liitännäisen rakenne

Koska lisätavoitteina on yksinkertaisuus ja helppokäyttöisyys, peliobjektien ja komponenttien serialisointi hoidetaan niin, että käyttäjän ei tarvitse määrittää jokaisen haluamansa komponentin kohdalla, miten se serialisoidaan. Tämä poissulkee esimerkiksi Uni-

tyn ISerializationCallbackReceiver-rajapinnan toteuttamisen, koska jokaisen serialisoitavan komponentin tulisi toteuttaa kyseinen rajapinta. Yksinkertaisimmaksi tavaksi toteuttaa tällainen serialisointi todettiin olevan Proxy-suunnittelumallia mukaillen. Proxy-suunnittelumallissa luodataan luokalle korvike, joka toimii rajapintana oikealle objektille. Proxy-luokkia käytetään GameObject-, Transform- ja Component-luokkien korvikkeena. Niiden avulla piilotetaan kentät, joita ei ole mahdollista serialisoida.

Jotta erilaisten komponenttien serialisointi ja liitännäisen konfigurointi olisi käyttäjälle yksinkertaista, serialisointi ja deserialisointi toteutetaan reflektion avulla. Reflektiolla tarkoitetaan kykyä muokata ohjelman toiminnallisuutta ja rakennetta ajoaikana. Liitännäisen tapauksessa reflektiota käytetään komponentin kenttien ja ominaisuuksien nimien ja arvojen selvittämiseen ja muokkaamiseen.

Liitännäisen helppokäyttöisyyden vuoksi liitännäisessä on yksi staattinen manageriluokka, jonka avulla käyttäjä pystyy luomaan ja lukemaan tallennustiedostoja. Kuvassa 3 näkyy liitännäisen luokkakaavio.



Kuva 3. Liitännäisen luokkakaavio

SaveManager-luokassa on funktiot CreateSave() ja LoadSave(), joiden avulla käyttäjä voi helposti tallentaa ja ladata Scenen tilan. Lisäksi SaveManager tarjoaa 4 muuta funktiota, jos käyttäjä haluaa itse hallita SceneData-luokan täyttämisen ja lukemisen tai Scenen tilan kirjoittamisen ja lataamisen.

5 Liitännäisen toteutus

5.1 Scenen serialisointi

Serialisoinnin kannalta keskeisimpänä luokkana toimii SaveManager. SaveManager on luokka, jonka funktioita käyttäjä voi kutsua tallentaakseen Scenen-nykyisen tilan tai ladatakseen aiemmin tallennetun tilan ja korvata nykyisen tilan ladatuilla komponenttien arvoilla. SaveManager noudattaa Singleton-suunnittelumallia, jolloin SaveManager-luokasta voi maksimissaan olla vain yksi ilmentymä. Lisäksi luokan ilmentymään pääsee kaikkialta käsiksi kutsumalla luokan staattista Instance-ominaisuutta. Jos SaveManager-luokasta ei ole vielä luotu ilmentymää Instance-ominaisuutta kutsuttaessa, SaveManager luo uuden peliobjektin, jossa on komponenttina SaveManager-skripti. Lisäksi SaveManager kutsuu Unityn funktiota DontDestroyOnLoad(), jolloin manageria ei tuhota, kun uusi Scene ladataan.

Jotta SaveManager-luokka tietäisi, mitkä peliobjektit sen tulee tallentaa, tarvitaan tietorakenne, joka pitää sisällään listan tallennettavista peliobjekteista. Lisäksi tallennetun peliobjektin tulee deserialisoitaessa muokata samaa peliobjektia, joten jokaisella tallennettavalla peliobjektilla on oltava uniikki tunniste. Dictionary on oiva tietorakenne tähän tarkoitukseen, mutta koska Unity ei tue Dictionary-luokan serialisointia, täytyy avain-arvo-pari-tyyppinen tietorakenne toteuttaa itse. Numero-peliobjekti-pari-tietorakennetta varten on luokka SaveablesDictionary, joka koostuu erillisistä avain- ja arvo-taulukoista. SaveablesDictionary-luokan tarkoituksena on siis pitää kirjaa Scenessä olevista peliobjekteista, jotka halutaan tallentaa.

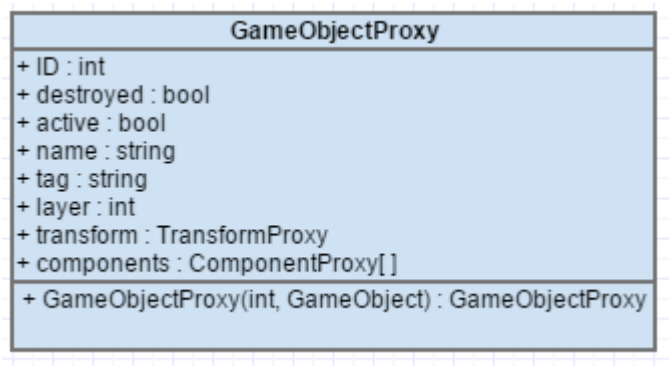
Yhdessä Scenessä voi korkeintaan olla yksi SaveablesDictionary. Taulukoiden sisällöt koostetaan pelikehitysympäristössä Scenen muokkausvaiheessa eikä sitä muokata ajoaikana ollenkaan. Vaihtoehtoisesti jokainen peliobjekti voisi itse lisätä omassa Awake()-funktiossaan GameObject-luokan SaveablesDictionary-luokkaan. Ennalta koostetun listan etuna on se, että Sceneä ladattaessa ei tarvitse huolehtia Awake()-kutsujen järjestyksestä.

Tallennettavat peliobjektit merkitään lisäämällä niihin Saveable-komponentti, joka pitää sisällään Scenekohtaisesti uniikin int-tyyppisen numerotunnisteen ja funktion, joka aset-

taa peliobjektin ja sen komponenttien arvot deserialisoitaessa tallennustiedostoa vastaaviin arvoihin. Käyttäjä voi lisätä Saveable-komponentin manuaalisesti tai hyödyntäen kehitysympäristölaajennusta, joka lisää komponentin kaikkiin liitännäisasetusten prefabejä vastaaviin peliobjekteihin.

Jotta peliobjektien ja niiden Transform-komponenttien serialisointi onnistuu valmista serialisointirajapintaa käyttäen, täytyy peliobjekteista ja Transform-komponenteista luoda korvikkeet. Komponenttien korvikkeen luonti on monimutkaisempaa, koska komponentteja voi olla monia erilaisia. Komponenteille on luotava yhteinen korvike, jonka avulla onnistuu sekä Unityn omien komponenttien, että käyttäjän itse luomien komponenttien serialisointi.

GameObject-luokan korvikkeena on GameObjectProxy-luokka, joka on kuvan 4 mukainen.

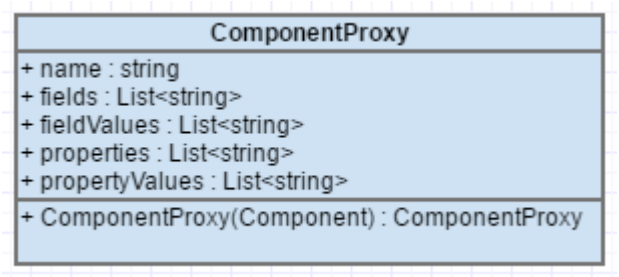


Kuva 4. GameObjectProxy-luokan rakenne

Unityn alkuperäisen GameObject-luokan kenttien lisäksi sen korvikkeessa on ID-, destroy- ja components-kenttä. Korvike luodaan kutsumalla sen konstruktoria käyttäen parametrina Saveable-komponentin ID-kenttää ja peliobjektia, johon kyseinen Saveable-komponentti kuuluu. Jos parametrina annetun peliobjektin arvo on null, destroyed-kentän arvoksi tulee true. Tämä tapahtuu, kun peliobjekti tuhotaan ja SaveableDictionaryssa oleva viittaus kyseiseen peliobjektiin katoaa.

Normaalisti GameObject-luokan transform-kentän tyyppi on Unity-pelimoottorin Transform. Sen tyyppi on peliobjektin korvikeluokassa TransformProxy. TransformProxy-luokka toimii Transform-luokan korvikkeena ja pitää sisällään peliobjektin sijainnin, kierron ja skaalan, aivan kuten alkuperäinenkin luokka.

Lisäksi GameObjectProxy-luokan konstruktori hakee peliobjektin kaikki komponentit ja luo jokaisesta komponentista ComponentProxy-korvikeluokan. ComponentProxy-luokka on kuvan 5 mukainen.



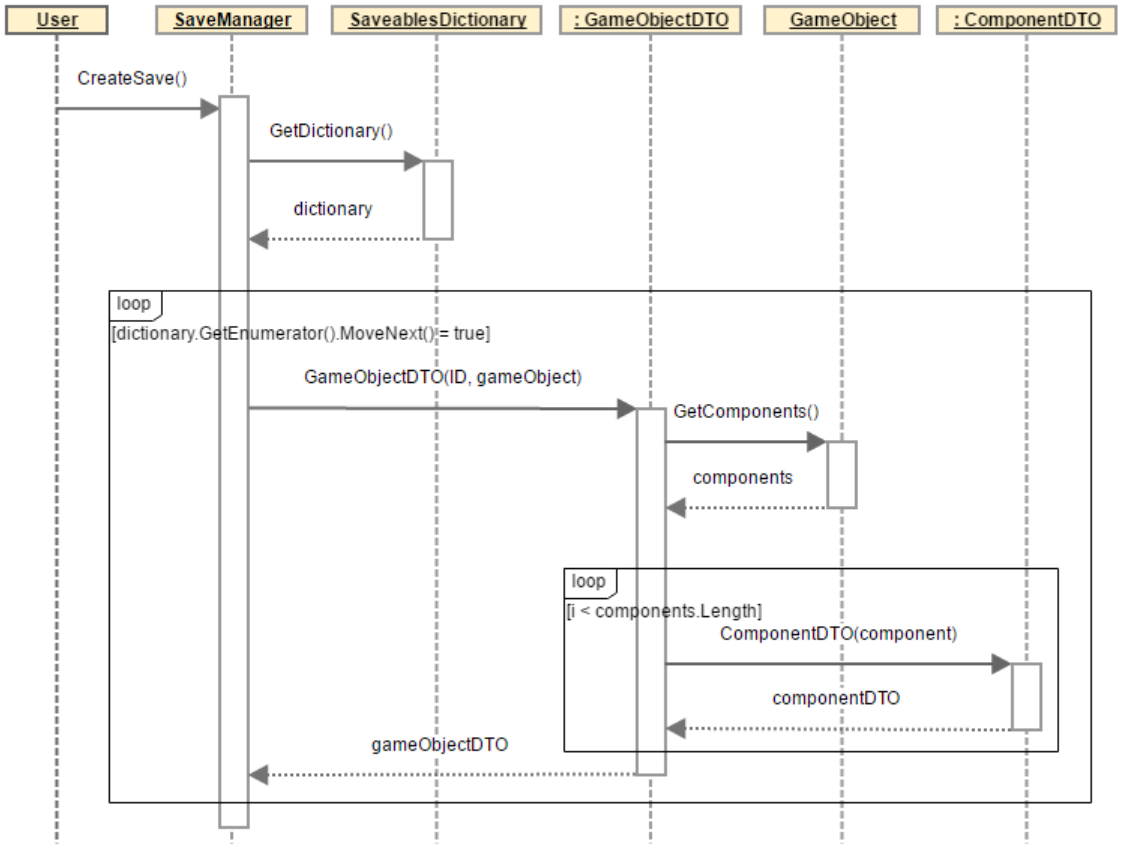
Kuva 5. ComponentProxy-luokan rakenne

ComponentProxy-luokan konstruktori saa parametrinaan komponentin, josta korvike luodaan. ComponentProxy-luokan name-kenttä on komponentin nimi, joka toimii myös komponentin tyyppinä. Konstruktori hakee reflektion avulla komponentin julkiset instanssimuuttujat ja -ominaisuudet (property). C#-ohjelmointikielessä reflektio onnistuu käyttäen .NET-ohjelmistokomponenttikirjaston System.Reflection-nimiavaruuden luokkia ja System.Type-luokkaa. Aluksi otetaan selville komponentin tyyppi kutsumalla sen GetType()-funktiota, jonka jälkeen kutsutaan tyyppimuuttujan GetFields()-funktiota. Näin päästään käsiksi komponentin kenttien nimiin ja arvoihin. Komponentin kenttien nimet lisätään fields-listaan ja arvot fieldValues-listaan string-tyyppisenä.

Komponentin ominaisuuksien selvittäminen on hieman monimutkaisempaa. Jotta ominaisuus voidaan deserialisoida, on varmistettava, että sen arvo voidaan asettaa, eli sillä on set-metodi. Komponentin ominaisuudet haetaan kutsumalla tyyppimuuttujan GetProperties()-funktiota, joka palauttaa PropertyInfo-taulukon. Kullekin taulukon PropertyInfo:lle kutsutaan GetSetMethod()-funktiota ja tarkistetaan set-metodin olevan olemassa ja julkinen. Ominaisuuksien nimet lisätään properties-listaan ja arvot propertyValues-listaan. Deserialisoinnin yksinkertaistamisen vuoksi liitännäinen serialisoi vain kentät ja ominaisuudet, joiden arvo on primitiivityyppinen.

Scenen nykyisen tilan kuvaamista ja serialisoinnin yksinkertaistamista varten tehdään tietorakenne (struct). SceneData-tietorakenne pitää sisällään Scenen nimen ja listan tallennettavien peliobjektien korvikkeista. Kun käyttäjä kutsuu SaveManagerin CreateSave()-funktiota, SaveManager täyttää kyseisen tietorakenteen parhaillaan aktiivisena

olevasta Scenestä löytyvän SaveablesDictionary-tietorakenteen avulla. SceneData-tietorakenteen name-kentälle haetaan arvo käyttäen Unityn SceneManagement-luokan GetActiveScene()-kutsua ja GameObjectProxy-lista luodaan kuvan 6 mukaisesti SaveablesDictionary-tietorakenteesta.



Kuva 6. Sekvenssikaavio SceneData-tietorakenteen täytöstä

SaveManager luo jokaisesta SaveablesDictionary-tietorakentessa olevasta peliobjektista GameObjectProxy-luokan ilmentymän. GameObjectProxy-luokan konstruktori hoitaa peliobjektin komponenttien korvikkeiden luonnin luomalla komponenteista ComponentProxy-listan. Kun GameObjectProxy-lista on luotu, SaveManager tekee SceneData-rakenteen ilmentymästä JSON-tiedoston Unityn tarjoamaa JsonUtility-rajapintaa käyttäen. Tiedosto tallennetaan käyttäjän liitännäisasetuksissa määrittämään tiedostopolkuun.

5.2 Tiedoston deserialisointi

Halutessaan ladata Scenen tilan tiedostosta, käyttäjä kutsuu SaveManagerin LoadSave()-funktiota. LoadSave()-funktio luo SceneData-tietorakenteen ja täyttää sen tallennustiedostossa olevien arvojen pohjalta. Sen jälkeen funktio käy läpi tietorakenteessa olevat elementit. Jos GameObjectProxy:n destroy-kentän arvo on true, tuhotaan Scenessä oleva korviketta vastaava peliobjekti. Muulloin kutsutaan Saveable-luokassa olevaa funktiota, joka hoitaa Scenessä olevan peliobjektin arvojen ylikirjoittamisen.

Peliobjektin kenttien korvaaminen on yksinkertainen uuden arvon sijoitus vanhan arvon tilalle. Komponenttien kenttien korvaaminen on monimutkaisempaa, koska kenttien nimet ovat yhdessä listassa ja arvot toisessa. Tästä johtuen komponenttien arvojen korvaamisessa on jälleen käytettävä reflektiota. Koska Saveable-komponentti on kiinni peliobjektissa, voidaan peliobjektille kutsua GetComponent(string)-funktiota antaen parametrina komponentin nimi, jolloin saadaan selville komponentin tyyppi. Kun tyyppi on tiedossa, voidaan reflektion avulla selvittää jokaisen kentän ja ominaisuuden tyyppi käyttäen komponentin kentän nimeä. Lisäongelmia aiheuttaa tallennustiedostossa olevien komponenttien kenttien ja ominaisuuksien arvot, koska ne ovat kaikki string-tyyppisiä. Koska komponenttien arvojen serialisointi on rajoitettu primitiivityyppeihin, on string-tyyppinen arvo helppo muuttaa oikean tyyppiseksi Convert-luokan ChangeType()-funktiolla.

Jotta komponentin kenttien ja ominaisuuksien arvot voidaan korvata, tarvitaan tieto varsinaisen komponentin kentistä ja ominaisuuksista. Pelkästään string-tyyppinen nimi ei riitä. Tieto kentistä saadaan selville kutsumalla komponentin tyypille GetFieldInfo(string)-funktiota ja vastaavasti ominaisuuksille GetPropertyInfo(string)-funktiota. Tämän jälkeen arvon asettaminen onnistuu helposti FieldInfo- tai PropertyInfo-luokan SetValue-funktiolla.

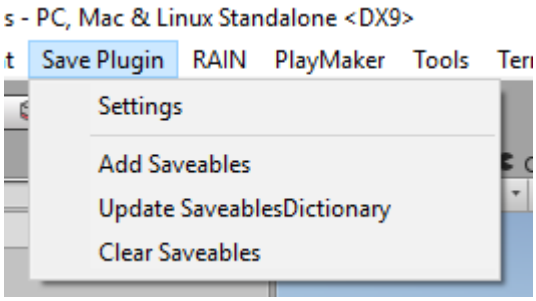
5.3 Käyttöliittymä

Unity-pelikehitysympäristöön voi luoda omia valikkoja ja näkymiä. Valikkojen lisääminen onnistuu luomalla staattisen funktion, jonka edessä on MenuItem-attribuutti. Kuvassa 7 näkyy esimerkki MenuItem-attribuutin käytöstä.

```
[MenuItem("Save Plugin/Settings", priority = 0)]
static void ShowWindow() {
```

Kuva 7. MenuItem-attribuutin käyttö oman valikon luomiseen

Attribuutin parametreina annetaan valikon polku ja halutessaan voi antaa myös valinnalle prioriteetin. Prioriteetilla 0 valinta on valikossa ylimpänä. Kuvassa 8 näkyy liitännäiselle luotu valikko.



Kuva 8. Liitännäisen valikko

Asetuksia varten luodaan ScripatbleObject-luokan perivä SavePluginSettings-luokka. Tällöin asetukset on helppo tallentaa kutsumalla AssetDatabase-luokan CreateAsset-funktiota. Kun asetukset tallennetaan ”Resources”-nimiseen kansioon, ne ovat myös helppo ladata pelin ajoaikana käyttäen Unityn Resources-luokkaa.

Käyttöliittymänä liitännäisen asetuksia varten parhaiten kehitysympäristölaajennuksista sopii erillinen näkymä. Näkymän luonti onnistuu perimällä EditorWindow-luokka ja toteuttamalla OnGUI()-funktio. Kun Settings-valintaa painetaan, eli ShowWindow()-funktiota kutsuttaessa, haetaan projektin resursseista SavePluginSettings.asset-tiedostoa

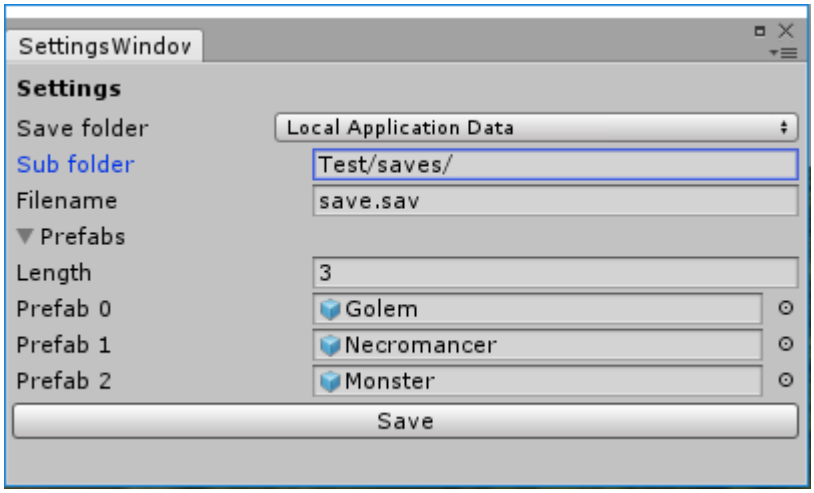
Resources.Load(path)-funktiolla. Jos resurssia ei ole olemassa, luodaan uusi SavePluginSettings.asset-tiedosto projektin ”Assets/Resources/”-kansioon.

Näkymään voi lisätä GUILayout- ja EditorGUILayout-luokkien funktioiden avulla erilaisia elementtejä. Kuvassa 9 on esimerkki tekstikentän lisäämisestä näkymään OnGUI()-funktiossa.

```
path = EditorGUILayout.TextField("Sub folder", path);
filename = EditorGUILayout.TextField("Filename", filename);
```

Kuva 9. Funktiokutsut, joilla saadaan tekstikenttä lisättyä näkymään.

TextField-funktion ensimmäinen parametri on tekstikentän nimike ja toinen parametri on kentässä näkyvä arvo. Kun tekstikenttää muokataan, funktio palauttaa muokatun string-muuttujan. Kuvassa 10 on näkymä, jossa pystyy muuttamaan liitännäisen asetuksia. Asetuksiin kuuluu tallennuskansio, tiedostonimi ja prefabit, joihin Saveable-komponentti lisätään.



Kuva 10. Liitännäisen asetusnäkymä

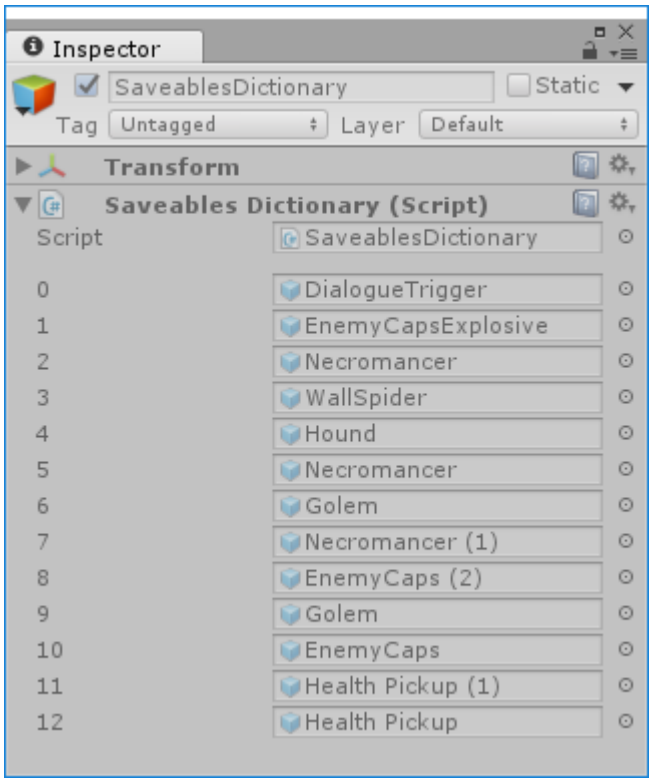
Asetukset tallennetaan painamalla tallennusnappulaa, jolloin näkymässä olevat arvot asetetaan SavePluginSettings-luokan arvoiksi, asetusresurssi merkitään muokatuksi SetDirty()-funktiolla ja tallennetaan takaisin AssetDatabaseen AssetDatabase.SaveAssets()-funktiolla. Asetusnäkymää avattaessa kenttien arvot täytetään asetusten nykyisillä arvoilla, jos asetukset on tallennettu aikaisemmin. Muulloin kenttiin sijoitetaan vakioarvot.

Asetusten lisäksi liitännäisen valikossa (kuva 8) on toimintoja, joilla käyttäjä pystyy helposti lisäämään Saveable-komponentteja asetusten mukaisista prefabeista luotuihin peliobjekteihin, päivittämään SaveablesDictionaryn sisällön, jos Saveable-komponentteja on lisätty manuaalisesti ja poistamaan kaikki Saveable-komponentit sekä SaveablesDictionaryn.

Kun valikosta valitaan ”Add Saveables”, toiminto tarkistaa, löytyykö parhaillaan aktiivisesta Scenestä SaveablesDictionary-luokkaa. Jos Scenestä sitä ei löydy, luodaan SaveablesDictionary-luokasta uusi ilmentymä Sceneen. Tämän jälkeen käydään läpi kaikki Scenessä olevat peliobjektit läpi ja tarkistetaan, ovatko ne peräisin prefabista. Tarkistus onnistuu tutkimalla liitännäisen asetuksissa olevaa listaa ja vertaamalla, vastaako mikään niistä Unityn PrefabUtility-luokan GetPrefabParent(GameObject)-funktion palauttama arvo mitään listan prefabeista. Jos peliobjektissa ei valmiiksi ole Saveable-komponenttia, lisätään siihen sellainen. Muulloin varmistetaan, että SaveablesDictionary-luokasta löytyy kyseinen peliobjekti. Aina tallennettavan peliobjektin löytyessä uniikkia tunnistetta kasvatetaan yhdellä.

Päivitys- ja poistovalintojen toiminnot ovat rakenteeltaan hyvin samanlaisia lisäysvalinnan toiminnon kanssa. Molemmat toiminnot prefabien etsimisen sijaan etsivät Scenessä olevat Saveable-komponentit. Komponenttien etsintä onnistuu helposti kutsumalla jokaiselle hierarkian juuripeliobjektille GetComponentsInChildren<T>(). Funktio palauttaa taulukon kaikista löytyneistä komponenteista, jotka ovat tyyppiä T. Päivitysvalinta täyttää SaveablesDictionary-luokan löytämiensä komponenttien perusteella, ja poistovalinta poistaa kaikki Saveable-komponentit peliobjekteista ja poistaa SaveablesDictionary-peliobjektin Scenestä.

Myös Inspector-näkymässä olevien komponenttien esitystä on mahdollista muokata. Jotta käyttäjä pystyisi helposti tutkimaan, missä kaikissa peliobjekteissa on Saveables-komponentti, toteutetaan SaveablesDictionary-luokalle mukautettu Inspector-näkymä. Mukautettu Inspector-näkymä tehdään luomalla SaveablesDictionaryEditor-luokka, joka perii Editor-luokan ja jolla on CustomEditor-attribuutti luokan edessä. Attribuutille annetaan parametrina tyyppimuuttuja, jonka tyyppisen luokan Inspector-näkymä halutaan toteuttaa. Lisäksi SaveablesDictionaryEditor-luokan on toteutettava OnInspectorGUI()-funktio. Kenttien lisääminen Inspector-näkymään onnistuu samalla tavalla kuin kenttien lisääminen asetusnäkymään GUILayout- ja EditorGUILayout-luokkien funktioiden avulla. Kuvassa 11 näkyy SaveablesDictionary-luokalle luotu Inspector-näkymä.



Kuva 11. SaveablesDirectory-peliobjektin Inspector-näkymä

6 Liitännäisen rajoitteet ja jatkokehitys

Liitännäisen avulla pystyy jo kattavasti tallentamaan paljon pelin tilasta. Hyvin paljon on kuitenkin parannettavaa sekä serialisoinnissa että käyttöystävällisyydessä. Suurin liitännäisen rajoite on se, että se ei kykene serialisoimaan komponenttien kaikkia serialisoitavia kenttiä. Pelkät primitiivityyppiset kattavat peruskäytön, mutta olisi erittäin hyödyllistä, jos liitännäinen serialisoisi myös esimerkiksi listoja ja luokkia, jotka muuten olisivat serialisoitavia.

Käyttöystävällisyyttä voitaisiin parantaa lisäämällä asetusnäkymään syötteen validointi. Tällä hetkellä liitännäinen ei tee minkäänlaisia tarkistuksia kenttien sisällöstä. Sen tulisi tarkistaa, että esimerkiksi käyttäjän syöttämä tiedostopolku ei sisällä kiellettyjä merkkejä.

Lisäksi liitännäinen ei tarkista, ovatko Scenen Saveable-komponenteissa olevat tunnistenumerot varmasti uniikkeja. Pelikehitysympäristölaajennuksen lisäystoiminto antaa komponenteille uniikin tunnistenumeron, mutta käyttäjä voi silti manuaalisesti käydä muuttamassa tunnistenumeroita tai lisätä Saveables-komponentin peliobjektiin ja antaa sille jo käytössä olevan tunnistenumeron. Lisäksi olisi hyvä pitää kirjaa kaikista käytetyistä tunnistenumeroista, jos kenttää muokataan pelin elinkaaren aikana. Eli jos kentästä julkaistaan ensin yksi versio, ja myöhemmin toinen versio, jossa kentästä on poistettu vihollinen, tulisi liitännäisen muistaa poistetun vihollisen tunnistenumero.

Tällä hetkellä liitännäinen pystyy tallentamaan vain yhden Scenen tilan. Käyttäjän pystyy kiertämään tämän rajoitteen luomalla uuden tallennustiedoston jokaista Sceneä kohti. Liitännäistä tullaan kehittämään niin, että liitännäinen serialisoi SceneData-listan kaikista Sceneistä, joiden ollessa aktiivisia on kutsuttu CreateSave()-funktio. Vaihtoehtoisesti asetuksiin voitaisiin lisätä asetus, jossa käyttäjä voi määrittää, mitkä Scenet serialisoidaan.

Muut jatkokehitysideat ovat pienemmän prioriteetin ideoita, ja ne tekevät liitännäisestä vähemmän yksinkertaisen käyttää, joka on alussa asetettujen tavoitteiden vastaista. Nämä kehitysideat kuitenkin tekisivät liitännäisestä entistä yleiskäyttöisemmän. Käyttäjälle voitaisiin antaa mahdollisuus valita, mitä serialisointiformaattia käytetään ja tuki tietokantaan tallennukselle voisi osoittautua myös tarpeelliseksi.

Tällä hetkellä liitännäinen serialisoi kaiken, mitä se serialisoitavasta peliobjektista löytää, jolloin tallennustiedosto sisältään erittäin paljon ylimääräistä dataa. Suuria määriä peliobjekteja serialisoitaessa voi myös huomata suorituskyvyssä pienen notkahduksen. Tämä johtuu useista reflektiokutsuista komponentteja serialisoitaessa. Suorituskykyä voisi optimoida rajoittamalla, mitä komponentteja serialisoidaan.

Asetusnäkymään voitaisiin lisätä prefabkohtaisia asetuksia, jossa käyttäjä voisi valita jokaisen prefabin kohdalla, mitkä komponentit ja mitkä komponenttien kentät serialisoidaan. Vaihtoehtoisesti näkymässä voisi olla lista, johon käyttäjä voi lisätä komponentteja. Näitä komponentteja ei serialisoitaisi minkään peliobjektin kohdalla. Lisäksi prefabien kohdalla voisi olla asetus, joka mahdollistaa myös peliobjektin lapsipeliobjektien serialisoinnin.

7 Yhteenveto

Insinöörityössä toteutettu liitännäinen täyttää pääosin projektia aloittaessa asetetut vaatimukset ja tavoitteet. Koska liitännäinen on vielä erittäin herkkä käyttäjän tekemille virheille, en koe liitännäisen vielä olevan valmis julkaistavaksi Unityn Asset Storessa. Tämän lisäksi julkaisua varten liitännäisen käyttöönotto tulisi olla helppoa käyttäjälle, joten liitännäinen dokumentoitava selkeästi ja liitännäisen mukana voisi tulla esimerkki Scene ja skriptit, jotka alustavasti demonstroisivat liitännäisen käytön.

Vaikka liitännäinen ei olekaan suorituskyvyltään optimaalinen, voidaan toteutukseen silti olla tyytyväisiä. Liitännäinen on yksinkertainen käyttää ja sen avulla pystyy tallentamaan moitteetta Scenet, joissa ei ole valtavaa määrää serialisoitavia komponentteja. Optimaalisinta olisi kuitenkin olla käyttämättä reflektiota lainkaan ja luoda korvikeluokka juuri niille komponenteille, jotka halutaan serialisoida. Kenties jos asetusnäkymään päätetään toteuttaa serialisoitavien komponenttien valinta, voidaan korvikeluokat mahdollisesti generoida kehitysympäristössä käyttäjän asetusten mukaisesti.

Unity oli ennestään tuttu kehitysympäristö, mutta liitännäisen toteuttaminen opetti paljon reflektiosta ja serialisoinnista. Ennen insinöörityötä reflektio oli tuttu vain käsitteenä. Insinöörityön aikana reflektio tuli erittäin tutuksi. Myös serialisoinnin hyödyntäminen ennen liitännäisen toteuttamista on ollut vähäistä. Insinöörityötä tehdessä tuli tutustuttua useisiin erilaisiin serialisointiformaatteihin ja -menetelmiin.

Lähteet

1 Newzoo Free 2016 Global Games Market Report. Verkkodokumentti. <http://resources.newzoo.com/hubfs/Reports/Newzoo_Free_2016_Global_Games_Market_Report.pdf>. 6.2016. Luettu 7.11.2016.

2 Unity – Fast Facts. Verkkodokumentti. <https://unity3d.com/public-relations>. Luettu 7.11.2016.

3 Brodkin, Jon. 2013. How Unity3D Became a Game-Development Beast <http://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/>. 3.6.2013. Luettu 7.11.2016.

4 Unity – Multiplatform. Verkkodokumentti. <https://unity3d.com/unity/multiplatform>. Luettu 7.11.2016.

5 Strandborg, Mikko. 2016. Introducing the Vulkan renderer preview. Verkkodokumentti. <https://blogs.unity3d.com/2016/09/29/introducing-the-vulkan-renderer-preview/>. 29.9.2016. Luettu 7.11.2016.

6 Anthony. 2014. High-performance physics in Unity 5. Verkkodokumentti. <https://blogs.unity3d.com/2014/07/08/high-performance-physics-in-unity-5/>. 8.7.2016. Luettu 7.11.2016.

7 Meijer, Lucas. 2010. Verkkodokumentti. <http://answers.unity3d.com/questions/9675/is-unity-engine-written-in-monoc-or-c.html>. 4.1.2010. Luettu 7.11.2016.

8 Serialization (C#). 2015. Verkkodokumentti. Microsoft. <https://msdn.microsoft.com/en-us/library/mt656716.aspx>. Päivitetty 20.7.2015. Luettu 7.11.2016.

9 JSON: The Fat-Free Alternative to XML. Verkkodokumentti. JSON. <http://www.json.org/xml.html>. Luettu 7.11.2016.

10 Novak, Maxim. Serialization Performance comparison (C#/.NET). Verkkodokumentti. <http://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserializer-xmlserializer-binaryformatter-json-newtonsoft-servicestack-text/>. Luettu 7.11.2016.

11 Dunstan, Jackson. 2015. More JSON Performance Benchmarks <http://jackson-dunstan.com/articles/3303>. 21.12.2015. Luettu 7.11.2016.